Capítulo NOVE

Expressões Lambda

Objetivos do exame:
Criar e usar expressões Lambda.

O que é uma expressão lambda?

No capítulo anterior, vimos como passar um trecho de código em vez de um objeto inteiro é muito útil.

O Java 8 introduziu uma notação que permite escrever:

```java
List compactCars = findCars(cars,
     (Car c) ->
        car.getType().equals(CarTypes.COMPACT);
```

Em vez de:

```java
List<Car> compactCars = findCars(cars,
     new Searchable() {
        public boolean test(Car car) {
           return car.getType().equals(
                     CarTypes.COMPACT);
        }
});
```

Estas são expressões lambda.

O termo “expressão lambda” vem do cálculo lambda (*lambda calculus*), escrito como λ-cálculo, onde λ é a letra grega lambda. Essa forma de cálculo trata da definição e aplicação de funções.

Com lambdas, você não será capaz de fazer coisas que não podia fazer antes delas, mas elas permitem que você programe de forma mais simples com um estilo chamado **programação funcional**, um paradigma diferente da programação orientada a objetos.

Uma Expressão Lambda

```java
(Object arg1, int arg2) -> arg1.equals(arg2);
```

Uma expressão lambda tem três partes:

Uma lista de parâmetros

Uma expressão lambda pode ter zero (representado por parênteses vazios), um ou mais parâmetros:

```java
() -> System.out.println("Hi");
(String s) -> System.out.println(s);
(String s1, String s2) -> System.out.println(s1 + s2);
```

O tipo dos parâmetros pode ser declarado explicitamente ou pode ser inferido do contexto:

```java
(s) -> System.out.println(s);
```

Se houver um único parâmetro, o tipo é inferido e não é obrigatório usar parênteses:

```java
s -> System.out.println(s);
```

Se a expressão lambda usar um nome de parâmetro que é o mesmo nome de uma variável no contexto envolvente, um erro de compilação é gerado:

```java
// Isto não compila
String s = ""; s -> System.out.println(s);
```

Uma seta (*arrow*)

Formada pelos caracteres - e > para separar os parâmetros do corpo.

Um corpo

O corpo das expressões lambda pode conter uma ou mais instruções.

Se o corpo tiver uma única instrução, as chaves não são necessárias e o valor da expressão (se houver) é retornado:

```java
() -> 4;
(int a) -> a * 6;
```

Se o corpo tiver mais de uma instrução, as chaves são obrigatórias e, se a expressão retornar um valor, ele deve ser retornado com uma instrução return:

```java
() -> {
    System.out.println("Hi");
    return 4;
}
(int a) -> {
    System.out.println(a);
    return a * 6;
}
```

Se a expressão lambda não retornar resultado, uma instrução return é opcional. Por exemplo, as seguintes expressões são equivalentes:

```java
() -> System.out.println("Hi");
() -> {
    System.out.println("Hi");
    return;
}
```

Como interfaces funcionais se relacionam com tudo isso?

A assinatura do método abstrato de uma interface funcional fornece a assinatura de uma expressão lambda (essa assinatura é chamada de **descritor funcional**).

Isso significa que, para usar uma expressão lambda, você precisa primeiro de uma interface funcional. Por exemplo, usando a interface do capítulo anterior:

```java
interface Searchable {
    boolean test(Car car);
}
```

Podemos criar uma expressão lambda que recebe um objeto Car como argumento e retorna um boolean:

```java
Searchable s = (Car c) -> c.getCostUSD() > 20000;
```

Neste caso, o compilador inferiu que a expressão lambda pode ser atribuída à interface Searchable apenas pela sua assinatura.

Na verdade, expressões lambda não contêm informação sobre qual interface funcional estão implementando. O tipo da expressão é deduzido do contexto em que a lambda é usada. Esse tipo é chamado de **tipo-alvo** (*target type*).

No exemplo anterior, como a expressão lambda está sendo atribuída à interface Searchable, a lambda deve assumir a assinatura de seu método abstrato. Caso contrário, um erro de compilação é gerado.

Se estivéssemos usando a lambda como argumento de um método, o compilador usaria a definição do método para inferir o tipo da expressão:

```java
class Test {
    public void find() {
        find(c -> c.getCostUSD() > 20000);
    }
    private void find(Searchable s) {
        // Aqui vai a implementação
    }
}
```

Por causa disso, a mesma expressão lambda pode ser associada a diferentes interfaces funcionais se elas tiverem assinaturas de métodos abstratos compatíveis. Por exemplo:

```java
interface Searchable {
    boolean test(Car car);
}
interface Saleable {
    boolean approve(Car car);
}
//...
Searchable s1 = c -> c.getCostUSD() > 20000;
Saleable s2 = c -> c.getCostUSD() > 20000;
```

Para referência, os contextos onde o tipo-alvo de uma expressão lambda pode ser inferido são:

- Uma declaração de variável
- Uma atribuição
- Uma instrução return
- Um inicializador de array
- Argumentos de método ou construtor
- Uma expressão condicional ternária
- Uma expressão de cast

Contudo, se você entender o conceito, não precisa memorizar esta lista.

Mas afinal, o que é tudo isso de programação funcional, descritores funcionais, etc.?

Uma expressão lambda é uma **função**. Isso é um pouco diferente dos métodos que conhecemos porque está realmente relacionado a uma função matemática.

Uma função matemática recebe algumas entradas e produz algumas saídas, mas **NÃO TEM EFEITOS COLATERAIS**, o que significa que, desde que a chamemos com os mesmos argumentos, ela sempre retorna o mesmo resultado.

Claro, no Java você nem sempre pode programar de forma puramente funcional (ou seja, sem nenhum efeito colateral), apenas em estilo funcional.

Portanto, em termos práticos, pode ser seguro pensar em expressões lambda como métodos (ou funções) anônimos, pois elas não têm nome, assim como classes anônimas.

Como expressões lambda se relacionam com classes anônimas?

Expressões lambda são uma **ALTERNATIVA** às classes anônimas, mas **não são a mesma coisa**.

Elas têm algumas semelhanças:

- Variáveis locais (variáveis ou parâmetros definidos em um método) só podem ser usadas se forem declaradas como final ou forem efetivamente finais.
- Você pode acessar variáveis de instância ou estáticas da classe envolvente.
- Elas não devem lançar mais exceções do que as especificadas na cláusula throws do método da interface funcional. Somente do mesmo tipo ou um supertipo.

E algumas diferenças significativas:

- Para uma classe anônima, a palavra-chave this se refere à própria classe anônima.
- Para uma expressão lambda, ela se refere à classe envolvente onde a lambda foi escrita.
- Métodos default de uma interface funcional **não podem ser acessados** de dentro de expressões lambda. Classes anônimas **podem**.
- Classes anônimas são compiladas como classes internas. Mas expressões lambda são convertidas em métodos static privados (em alguns casos) da sua classe envolvente e, usando a instrução invokedynamic (adicionada no Java 7), são ligadas dinamicamente. Como não há necessidade de carregar outra classe, expressões lambda são mais eficientes. Essa é uma explicação simples, mas essa é a ideia.

Pontos-chave

- Expressões lambda têm três partes: uma lista de parâmetros, uma seta e um corpo:

```java
(Object o) -> System.out.println(o);
```

- Você pode pensar em expressões lambda como métodos (ou funções) anônimos, pois elas não têm um nome.
- Uma expressão lambda pode ter zero (representado por parênteses vazios), um ou mais parâmetros.
- O tipo dos parâmetros pode ser declarado explicitamente, ou pode ser inferido do contexto.
- Se houver um único parâmetro, o tipo é inferido e não é obrigatório usar parênteses.
- Se a expressão lambda usar um nome de parâmetro que seja o mesmo de uma variável no contexto envolvente, ocorre erro de compilação.
- Se o corpo tiver uma única instrução, chaves não são necessárias e o valor da expressão (se houver) é retornado.
- Se o corpo tiver mais de uma instrução, chaves são obrigatórias e, se a expressão retornar um valor, ele deve ser retornado com uma instrução return.
- Se a expressão lambda não retorna um resultado, uma instrução return é opcional.
- A assinatura do método abstrato de uma interface funcional fornece a assinatura de uma expressão lambda (essa assinatura é chamada de **descritor funcional**).
- Isso significa que, para usar uma expressão lambda, você precisa primeiro de uma interface funcional.
- No entanto, expressões lambda não contêm a informação sobre qual interface funcional estão implementando.
- O tipo da expressão é deduzido do contexto em que a lambda é usada. Esse tipo é chamado de **tipo-alvo** (*target type*).
- Os contextos onde o tipo-alvo de uma expressão lambda pode ser inferido incluem uma atribuição, argumentos de métodos ou construtores, e uma expressão de cast.
- Assim como classes anônimas, expressões lambda podem acessar variáveis de instância e estáticas, mas somente variáveis locais final ou efetivamente finais.
- Além disso, elas não podem lançar exceções que não estejam definidas na cláusula throws do método da interface funcional.
- Para uma expressão lambda, this se refere à classe envolvente onde a lambda foi escrita.
- Métodos default de uma interface funcional não podem ser acessados de dentro de expressões lambda.

Autoavaliação (*Self Test*)

1. Quais das seguintes são expressões lambda válidas?

A. String a, String b -> System.out.print(a + b);

B. () -> return;

C. (int i) -> i;

D. (int i) -> i++; return i;

---

2. Dado:

```java
interface A {
    int aMethod(String s);
}
```

Quais das seguintes são instruções válidas?

A. A a = a -> a.length();
B. A x = y -> { return y; };
C. A s = "2" -> Integer.parseInt(s);
D. A b = (String s) -> 1;

---

3. Uma expressão lambda pode ser usada...

A. Como argumento de método
B. Como expressão condicional em uma instrução if
C. Em uma instrução return
D. Em uma instrução throw

---

4. Dado:

```java
() -> 7 * 12.0;
```

Qual das seguintes interfaces pode fornecer o descritor funcional para a expressão lambda acima?

A.

```java
interface A {
    default double m() {
        return 4.5;
    }
}
```

B.

```java
interface B {
    Number m();
}
```

C.

```java
interface C {
    int m();
}
```

D.

```java
interface D {
    double m(Integer... i);
}
```

---

5. Dado:

```java
interface AnInterface {
    default int aMethod() { return 0; }
    int anotherMethod();
}
public class Question_9_5 implements AnInterface {
    public static void main(String[] args) {
        AnInterface a = () -> aMethod();
        System.out.println(a.anotherMethod());
    }

    @Override
    public int anotherMethod() {
        return 1;
    }
}
```

Qual é o resultado?

A. 0
B. 1
C. A compilação falha
D. Uma exceção ocorre em tempo de execução

---

6. Quais das seguintes afirmações são verdadeiras?

A. Chaves são obrigatórias sempre que a palavra-chave return for usada em uma expressão lambda
B. A palavra-chave return é sempre obrigatória em uma expressão lambda
C. A palavra-chave return é sempre opcional em uma expressão lambda
D. Expressões lambda não retornam valores

---

7. Como a palavra-chave this é tratada dentro de uma expressão lambda?

A. Você não pode usar this dentro de uma expressão lambda
B. this se refere à interface funcional da expressão lambda
C. this se refere à própria expressão lambda
D. this se refere à classe envolvente da expressão lambda

---

8. Dado:

```java
interface X {
    int test(int i);
}
public class Question_9_8 {
    int i = 0;
    public static void main(String[] args) {
        X x = i -> i * 2;
        System.out.println(x.test(3));
    }
}
```

Qual é o resultado?

A. 0
B. 3
C. 6
D. A compilação falha